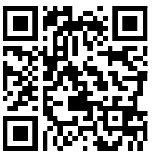
软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(6):1703−1722 [doi: 10.13328/j.cnki.jos.005847]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于堆叠泛化的设计模式检测方法*



冯 铁[1,3], 靳 乐[2], 张家晨[1], 王洪媛[1]

[1](吉林大学 计算机科学与技术学院,吉林 长春 130012)

[2](吉林大学 软件学院,吉林 长春 130012)

[3](符号计算与知识工程教育部重点实验室(吉林大学),吉林 长春 130012)

通讯作者: 王洪媛, E-mail: hongyuan@jlu.edu.cn

摘 要: 设计模式检测是理解和维护软件系统的一项重要工作.以高效识别设计模式变体和提高设计模式识别准确率为目的,将面向对象度量与模式微结构相结合,提出一种基于堆叠泛化的设计模式检测方法.该方法应用典型的机器学习算法,分别训练度量分类器和微结构分类器,并基于两者的预测值和相关对象模型特征进一步训练,从而形成堆叠分类器.为了评估该方法,基于该方法开发了一个原型工具 OOSdpd.该工具从 Java 字节码级别的系统实现中抽取设计模式实例,并在 JUnit 等几个经典的开源项目上进行实验.通过与现有的两种工具进行对比分析,实验验证了该方法在提高设计模式识别准确率及召回率方面的有效性.

关键词: 设计模式检测;面向对象软件度量;模式微结构;堆叠泛化;机器学习

中图法分类号: TP311

中文引用格式: 冯铁,靳乐,张家晨,王洪媛.基于堆叠泛化的设计模式检测方法.软件学报,2020,31(6):1703−1722. http://www.jos.org.cn/1000-9825/5847.htm

英文引用格式: Feng T, Jin L, Zhang JC, Wang HY. Design pattern detection approach based on stacked generalization. Ruan Jian Xue Bao/Journal of Software, 2020,31(6):1703−1722 (in Chinese). http://www.jos.org.cn/1000-9825/5847.htm

Design Pattern Detection Approach Based on Stacked Generalization

FENG Tie[1,3], JIN Le[2], ZHANG Jia-Chen[1], WANG Hong-Yuan[1]

[1](College of Computer Science and Technology, Jilin University, Changchun 130012, China)

[2](College of Software, Jilin University, Changchun 130012, China)

[3](Key Laboratory of Symbolic Computation and Knowledge Engineering (Jilin University), Ministry of Education, Changchun 130012, China)

Abstract: Design pattern detection plays an important role in understanding and maintaining software system. With the purpose of identifying variants of design pattern efficiently and improving the accuracy of design pattern detection, an approach of design pattern detection based on stacked generalization in combination with object-oriented software metrics and pattern micro-structures is proposed in this study. Applying some typical machine learning algorithms, the approach trains a metric classifier and a micro-structure classifier for each design pattern, after which a stacked classifier is further trained and constructed on the predictive values of the two classifiers and some related object modeling features. To evaluate the proposed approach, a prototype tool, namely OOSdpd, is developed to detect design pattern instances from Java bytecode files of a system. The experiments on several classic open source projects are carried out, such as JUnit etc., and the proposed approach is compared with two existing tools. Experiments prove the effectiveness of the proposed approach in terms of improving the accuracy and recall rate of design pattern detection.

Key words: design pattern detection; object-oriented software metric; pattern micro-structure; stacked generalization; machine learning

* 基金项目: 国家自然科学基金(61471181); 赛尔网络下一代互联网技术创新项目(NGII20180701)
Foundation item: National Natural Science Foundation of China (61471181); CERNET Innovation Project (NGII20180701)
收稿时间: 2018-04-27; 修改时间: 2018-09-20, 2018-12-06; 采用时间: 2019-03-29

设计模式通过标识对象、对象间的合作及责任分配来揭示设计机理,表明基于某种经验的、在特定的上下文中解决一个普遍设计问题的可复用体系结构[1,2].设计模式已经被广泛地应用在各种软件和工具库的设计与实现中.设计模式检测旨在从现有的软件设计文档或源代码当中识别所使用的设计模式实例,它不仅对于帮助维护人员理解软件系统的设计动机和原理、恢复软件设计和改善软件的可维护性具有重要意义,而且有助于软件体系结构的恢复和发现,同时也是评估软件质量的一个重要依据.

由于设计模式的应用大多是基于意图的,并未严格限制模式的具体实现,加之不同开发者对于设计模式的理解不一,这使得模式的实现千变万化.目前,设计模式检测主要存在以下问题:(1) 变体的检测效果不理想;(2) 结构相同意图不同的模式难以区分;(3) 行为型设计模式的检测复杂;(4) 组合爆炸问题依然突出.正是因为这些问题的存在,设计模式检测仍然是软件工程领域的一个重要的研究内容.

机器学习是人工智能领域的重要分支,它是使用经验来提高计算性能或做出准确预测的计算方法[3].机器学习技术适用于设计模式检测的原因如下:首先,设计模式检测活动本身就是对众多候选模式实例进行预测的过程,这符合机器学习的应用背景;其次,设计模式变体的检测与机器学习无需硬编码的特点相适应;最后,机器学习对语义信息的描述,有助于区分结构相同而意图不同的模式.

对于行为型设计模式的检测,目前主要的方法是借助于动态分析[4−6],但是动态分析一般要求目标项目完整可运行,这在实际中有时候是不可行的;另外,监控候选模式实例的运行需要生成相应的测试用例,如果手工生成,必然费时费力,如果自动生成,又难以做到有效的路径覆盖.如果能够充分利用静态代码分析技术,将对模式的检测研究深入到方法内部,并尝试用机器学习的方式去刻画模式的行为特征,就可以较大程度地降低行为型设计模式检测的难度.

本文提出一种新的设计模式检测方法,结合面向对象度量和模式微结构(以下简称度量和微结构)并使用典型的机器学习算法来实现设计模式的检测.针对每种设计模式,本文分别用度量和微结构(micro-structure,简称MS)训练出一个分类器,然后采用模型堆叠的方式训练出最终的分类器,从而实现对该设计模式的检测.

本文的主要贡献如下:(1) 提出了用多视图(设计模式度量视图和设计模式微结构视图)堆叠泛化的方法来解决设计模式检测问题;(2) 定义了模式必备微结构来缩减搜索空间,从而避免组合爆炸问题;(3) 对于检测出的模式实例,除给出角色映射外,还能给出具体的方法映射.最终实验表明,本文提出的方法在变体的识别、行为型设计模式的检测以及组合爆炸问题的解决等方面均有明显提升.

本文第 1 节介绍本文相关工作.第 2 节定义度量和微结构以及相关的基本概念与原理.第 3 节详细介绍基于堆叠泛化的设计模式检测方法.第 4 节在 5 种设计模式上进行实验,通过实验数据的对比分析,验证所提方法的有效性.第 5 节对本文的工作进行总结并对未来工作展望.

1 相关工作

自从 GoF 等人提出软件设计模式概念以来[1],许多学者对设计模式的自动化检测进行了研究,提出了很多种设计模式检测方法.这些方法在所用技术、分析类型、所面向的源代码或模式的中间表示、是否精确匹配、是否完全自动化、是否面向特定模式、实验的信息项和指标等方面各不相同.接下来,本文从所用技术方面对一些具有代表性的方法进行归纳和介绍.

有些设计模式检测方法是基于相似度计算的[7,8].例如,Tsantalis 等人利用图的相似性算法实现了设计模式的检测[7],该方法不仅能够识别模式变体,而且还能够有效地解决模式检测过程中的组合爆炸问题(利用了设计模式大都包含继承层次这一事实).简单来说,该方法将设计模式角色之间的各种关系用邻接矩阵来表示,然后从源码中提取所有的继承层次,接着在模式查找阶段,用继承层次中的类去映射设计模式的各个角色,然后提取出这些角色类之间的各种关系矩阵,最后利用矩阵之间的相似性算法得到一个最终的相似度,如果相似度超过了事先选择的阈值,那么就找到了一个设计模式实例.该方法的缺点是对某些行为型设计模式的检测效果不好,同时空间复杂度也很高.

有些设计模式检测方法是基于图理论的[9−11].例如,Yu 等人从设计模式的子模式出发,将设计模式检测问题

映射为图论中的图同构问题[9].他们定义了十几种能够表示设计模式结构特征的子模式.对于一个待检测系统,他们首先将系统源码和子模式用类关系有向图来表示,然后从系统源码的类关系有向图中找到和子模式的类关系有向图同构的子图,通过参照设计模式结构特征模型来组合这些子图,从而得到候选模式实例.为了追踪设计模式行为方面的特征,他们使用方法签名模板来过滤不符合条件的候选模式实例.他们方法的缺点是没有对子模式进行过滤,从而子模式挖掘和子模式组合的时间复杂度很高.另外,他们仅仅使用方法签名来追踪设计模式的行为特征,这显然是不够的.

还有一些设计模式检测方法是基于本体的[12−14].例如,Dietrich 等人使用 Web 本体语言(OWL)来形式化地定义设计模式和其相关概念[12].他们开发了一款原型工具,该工具以设计模式的形式化描述和待检测的 Java 项目为输入,输出待检测项目中该设计模式的实例.由于该方法采用的是准确匹配,因此很难对设计模式的变体进行检测.最后,他们提出可以去掉或弱化某些约束,从而实现模糊匹配,但是他们并未针对具体模式说明哪些约束可以被去掉或弱化.

最后,基于机器学习的设计模式检测方法也有很多[15−17],其中具有代表性的检测方法是下面 3 个:

Alhusain 等人首次尝试了只使用机器学习的方法来实现设计模式的检测[15].他们的训练数据集是由 MARPLE[18],SSA[7],WOP[12],FINDER[19]等 4 种模式检测工具投票产生的.他们的模式检测方案包括两个阶段:在第 1 阶段,他们通过特征选择算法为每个模式角色选择一个特征子集,然后根据该特征子集和相应的分类模型来确定每个模式角色的候选类,从而减小了模式的搜索空间;在第 2 阶段,每种设计模式都有一个单独的分类器,该分类器使用表示角色类之间关系的特征作为输入.最后他们在 JHotDraw 项目上进行了实验,但是结果并不是很理想.

Zanoni 等人也将机器学习技术应用到了设计模式检测当中,他们开发了一个名叫 MARPLE 的 Eclipse 插件,该插件可以从 Java 源代码当中抽取设计模式[16].MARPLE 插件包括 3 个主要模块:(1) 信息检测引擎,负责构建系统模型;(2) Joiner,负责从系统模型里面抽取设计模式候选实例;(3) Classifier,负责分类 Joiner 的结果,把候选实例分类为正确的模式实例和错误的模式实例.他们为每种设计模式都找到了最佳的聚类算法和分类算法.他们方法的缺点是训练数据集需要人工产生,而人工产生数据集会受主观因素影响;Joiner 的召回率并不是 100%;有效性和训练集大小的关系有待进一步验证;并未考虑系统库和第三方库中的代码.

Chihada 等人从面向对象度量的角度给出了设计模式的检测方法[17].他们的检测方法分为两个阶段:设计模式组织阶段和设计模式检测阶段.在设计模式组织阶段,他们从专家那里获取设计模式实例,然后通过实例计算特征向量,最终生成训练数据集,并学习出分类器.在设计模式检测阶段,它们首先补充源码图(由类图转化而来),也即让子类继承父类的各种关系,然后从源码图中枚举出候选实例,并从中过滤掉不包含抽象类或接口类的实例,然后用第 1 阶段的分类器进行分类,从而判断候选实例是不是真正的模式实例.该方法的缺点是只考虑了设计模式的度量信息.

总体来讲,上述工作在一定程度上解决了设计模式检测问题,但或多或少存在前面提到的 4 个问题.因此本文从这些问题出发,结合度量和微结构,提出了一种新的机器学习检测方法.该方法不仅能够提高变体的检测效果,而且对组合爆炸问题的解决也有很大贡献.同时,该方法尝试从微结构的角度去刻画模式的行为特征,对于行为型设计模式的检测效果也有明显提升.

2 面向对象度量与微结构

本节给出基于堆叠泛化的设计模式检测方法相关的两个概念(面向对象度量和微结构)的定义及表示方式,并列出了本文所使用的度量列表和微结构列表,着重定义了 5 种方法类微结构以及相应的检测方法.

2.1 面向对象度量

通过有效的特征选择算法为每种设计模式找到合适的面向对象度量特征集,是基于堆叠泛化的设计模式检测方法的重要步骤之一.面向对象度量通过定量地估算面向对象系统的设计特性,如类的继承深度、对象间耦合、方法内聚程度等,来预测测试的复杂性、发现系统中的错误和促进软件的模块化,它在一定程度上衡量

了软件的设计质量.另一方面,设计模式是设计经验的总结和提炼,设计模式实例的应用是改善设计质量的一个有效手段,从而针对设计模式实例的相关度量值也会呈现出一定的规律性.目前已经有许多学者利用面向对象度量进行了设计模式的检测研究[17,20−22],这些度量包括 C&K 度量[23]、Lorenz 度量[24]、Tegarden 度量[25]、Hitz &Montazeri 度量[26]以及 Briand 度量[27]等.面向对象度量可以分为属性级别的度量、方法级别的度量、类级别的度量(以下简称类度量)和系统级别的度量,用于设计模式检测的主要是类级别的度量.

通过对 GoF 的 23 个设计模式的观察和理解,表 1 所示的面向对象度量在刻画和标识设计模式实例方面具有重要作用,因此本文选择表 1 所示的 69 种类度量并通过 POM Tests 工具[21]来计算相应度量值.

Table 1 Metrics used in this paper

表 1 本文选取的度量列表

ACAIC	DCAEC	LCOM5	NMDExtended	NOTC	TLOC
ACMIC	DCC	LOC	NMI	NOTI	TestCaseLOC
AID	DCMEC	MFA	NMO	NPrM	USELESS
ANA	DIT	MLOCsum	NOA	PIIR	VGsum
CAM	DSC	MOA	NOC	PP	WMC
CBO	EIC	McCabe	NOD	REIP	WMC1
CBOin	EIP	NAD	NOF	RFC	WMC_New
CBOout	FanOut	NADExtended	NOH	RFC_New	cohesionAttributes
CIS	ICHClass	NCM	NOM	RFP	connectivity
CLD	IR	NCP	NOP	RPII	
CP	LCOM1	NMA	NOPM	RTP	
DAM	LCOM2	NMD	NOParam	SIX	

举例来讲,AID 表示一个类的平均继承深度(考虑多继承),它可以用来衡量一个类的抽象程度.一般而言,设计模式中的抽象角色类都具有较小的 AID 值.CBO 表示一个类和其他类之间的耦合程度,CBOin 表示入耦合,CBOout 表示出耦合,设计模式是设计经验的总结和提炼,它的角色类之间一般不会有太高的耦合程度.NOC 表示一个类的直接子类个数,NOC 越大,则代码的重用越好,但是抽象性减弱.

定义 1(类度量表示(class metric representation),简称 CMR). 类度量表示由一个三元组构成:*CMR*=(*classA*,*metric*,*value*).其中,*classA* 表示某个具体的类,*metric* 表示某个类度量,*value* 表示 *classA* 的 *metric* 度量值.

2.2 微结构

设计模式微结构是指类或对象之间的静态或动态关系,作为设计模式的组成部分,它具有更小的粒度,每个设计模式的意图可以通过组合微结构的意图进行表达.定义合理的特征选择算法为每种设计模式找到微结构特征集,是基于堆叠泛化的设计模式检测方法的另一个重要步骤.Fontana 等人认为,子模式[28]、元素模式[29]、微模式[30]以及他们自己所提出的设计模式线索[31]都属于设计模式微结构范畴,并且也证实了微结构和设计模式检测之间的密切关系[32].随后,他们基于微结构给出了一种新的设计模式检测方法[16].不过,虽然提出了微结构这一术语,但却没有给出明确的定义.同时,他们也认为现有的微结构对设计模式检测来讲并不是完备的.本文将对微结构进行了明确定义,并提出几种新的微结构.

微结构可以分为两种:方法类微结构和非方法类微结构.其中:方法类微结构描述设计模式中包含的方法方面的信息,比如子模式中的 Overriding Method[28]、元素模式中的 Create Object[29]以及设计模式线索中的 Multiple Redirections in Family[31]等;而非方法类微结构描述设计模式的角色类以及相关属性方面的信息,如 Association,Inheritance,Private Self Instance 等.

定义 2(微结构). 表示类或对象之间的静态或动态的关系,每个微结构由三元组表示:*MS*=(*classA*/*instance*, *classB*/*instance*,*relationalMask*).其中,*classA*/*instance* 和 *classB*/*instance* 表示该微结构的两个参与类或对象,而 *relationalMask* 表示它们之间的关系 *Mask* 值.对于方法类微结构,它是一个对应于 *classA*/*instance* 的比特向量,如果 *classA*/*instance* 的第 *n* 个方法和 *classB*/*instance* 之间体现了这种微结构关系,那么 *relationalMask* 的相应位置置 1;对于非方法类微结构,若 *classA*/*instance* 和 *classB*/*instance* 存在这种微结构关系,则 *relationalMask* 置 1.

为解决目前大多数基于机器学习的设计模式检测方法只能给出角色映射,而不能给出具体方法映射的问

题,在定义方法类微结构时,本文采用比特向量记录相关微结构出现的位置和数目.一旦找到一个设计模式实例,就可以通过相关微结构元组中的比特向量以及它们之间的位运算来找到对应的方法映射.

图 1 所示的 Java 代码中包含了一个对象适配器模式的实例,其中,Contact 接口类扮演对象适配器模式的 Target 角色,Employee 类扮演对象适配器模式的 Adaptee 角色,ContactAdapter 类扮演对象适配器模式的 Adapter 角色.该模式实例中存在一个 Overriding Method 微结构(ContactAdapter,Contact,01011).其中,01011 表示 ContactAdapter 的第 2、第 4 和第 5 个方法是对 Contact 的覆写.该模式实例中还存在另一个 Delegation 微结构(ContactAdapter,Employee,01010).其中,01010 表示 ContactAdapter 的第 2 和第 4 个方法是对 Employee 的委托.通过对这两个微结构中的比特向量做与运算,也即 01011&01010=01010,就能找到该对象适配器模式的方法映射.与运算值 01010 表示 ContactAdapter 的第 2 和第 4 个方法对应对象适配器模式的 *Request*(·)方法.

```
interface Contact {
   public String getName();
   public String getPhoneNumber();
   public String toString();
}
class ContactAdapter implements Contact {
   private Employee emp;
   private String title;
   public ContactAdapter(Employee employee) {
      emp = employee;
   }
   @Override
   public String getName() {
      return emp.getFirstName()+" "+emp.getLastName();
   }
   public void setTitle(String newTitle) {
      title = newTitle;
   }
   @Override
   public String getPhoneNumber() {
      return emp.getTelephoneNumber();
   }
   @Override
   public String toString() {
      return "Name: "+getName()+". PhoneNumber: "
                  +getPhoneNumber();
   }
}
```

```
class Employee {
   private String firstName;
   private String lastName;
   private String telephoneNumber;
   private String age;
   public Employee(String firstName,
                   String lastName,
                   String telephoneNumber,
                   String age) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.telephoneNumber = telephoneNumber;
      this.age = age;
   }
   public String getFirstName() {
      return firstName;
   }
   public String getLastName() {
      return lastName;
   }
   public String getTelephoneNumber() {
      return telephoneNumber;
   }
   public String getAge() {
      return age;
   }
}
```

Fig.1 An instance of object adapter pattern

图 1 一个对象适配器模式的实例

定义 3(微结构的维度(dimension of micro-structure)). 指定微结构所关联的类或对象的个数.对于定义 2 所定义的微结构,如果 *classA*/*instance* 与 *classB*/*instance* 相同,则称该微结构的维度为 1;否则,称该微结构的维度为 2.

定义 4(返回类型微结构(return type)). 如果类 *A* 中存在一个方法 *m*,*m* 的返回类型为类 *B*,则称类 *A* 到类 *B* 存在返回类型微结构.返回类型微结构可以表示为三元组(*classA*,*classB*,*ReturnTypeMask*),其中,*ReturnTypeMask* 是对应于 *classA* 的比特向量,如果 *classA* 的第 *n* 个方法的返回类型为 *classB*,那么 *ReturnTypeMask* 的相应位置置 1.显然,比特向量 *ReturnTypeMask* 的基数就是 *classA* 到 *classB* 的返回类型微结构的数目.

定义 5(多通知微结构(multi-notify)). 如果类 *A* 中存在某个包含循环的方法 *m*,并且循环体内存在对类 *B* 的返回类型为空的方法的调用,则称类 *A* 到类 *B* 存在多通知微结构.多通知微结构可以表示为三元组(*classA*,*classB*,*MultiNotifyMask*),其中,*MultiNotifyMask* 是对应于 *classA* 的比特向量.如果 *classA* 的第 *n* 个方法的内部有一个循环,并且循环体内有对 *classB* 的返回类型为空的方法的调用,那么 *MultiNotifyMask* 的相应位置置 1.

定义 6(参数化通知微结构(parameterized notify)). 如果类 *A* 的某个方法 *m* 内存在对类 *B* 的带参数且返回类型为空的方法的调用,则称类 *A* 到类 *B* 存在参数化通知微结构.参数化通知微结构可以表示为三元组(*classA*,*classB*,*ParameterizedNotifyMask*),其中,*ParameterizedNotifyMask* 是对应于 *class A* 的比特向量.如果 *classA* 的第 *n* 个方法内有对类 *B* 的带参数且返回类型为空的方法的调用,那么 *ParameterizedNotifyMask* 的相应位置置 1.

定义 7(参数依赖微结构(parameter dependence)). 如果类 *A* 存在某个方法 *m*,*m* 的参数表中包含类型为类 *B* 的参数,则称类 *A* 到类 *B* 存在参数依赖微结构.参数依赖微结构表示为三元组:

(*classA*,*classB*,*ParameterDependenceMask*).

其中,*ParameterDependenceMask* 是对应于 *classA* 的比特向量,如果 *classA* 的第 *n* 个方法的参数表中包含 *classB* 类型的参数,那么 *ParameterDependenceMask* 的相应位置置 1.

定义 8(返回静态自身实例微结构(static self instance returned)). 如果类 *A* 存在某个方法 *m*,*m* 的返回类型为 *static classA*,则称类 A 存在返回静态自身实例微结构.显然,该微结构是针对单个类而言的 1 维微结构.返回静态自身实例可以表示为三元组:

(*classA*,*classA*,*StaticSelfInstanceReturnedMask*).

其中,*StaticSelfInstanceReturnedMask* 是对应于 *classA* 的比特向量.如果 *classA* 的第 *n* 个方法的返回类型为 *static classA*,那么 *StaticSelfInstanceReturnedMask* 的相应位置置 1.

为了在工厂方法模式、单例模式、对象适配器模式、组合模式以及观察者模式上进行实验,本文初步选用了 16 种微结构,具体情况见表 2.

Table 2 Micro-structures used in this paper

表 2 本文选取的微结构列表

微结构	中文名	分类	维度
Overriding method	覆写方法	方法类	2
Create object	创建对象	方法类	2
Return type	返回类型	方法类	2
Delegation	委托	方法类	2
Multiple redirections in family	多重定向至父类	方法类	2
Redirect in family	重定向至父类	方法类	2
Multi-notify	多通知	方法类	2
Parameterized notify	参数化通知	方法类	2
Parameter dependence	参数依赖	方法类	2
Association	关联	非方法类	2
Protected instantiation	受保护实例化	方法类	1
Private self instance	私有自身实例	非方法类	1
Static self instance	静态自身实例	非方法类	1
Single self instance	单一自身实例	非方法类	1
Controlled self instantiation	受控自身实例化	方法类	1
Static self instance returned	返回静态自身实例	方法类	1

需要说明的是:本文微结构的检测是在 Java 字节码级别上进行的,检测工具使用了 ASM 框架(一个 Java 字节码操作和分析框架).

3 设计模式检测方法

本节首先给出该方法的总体框架,接着定义一些基本概念,然后利用定义好的概念对设计模式检测过程中的算法进行形式化描述,最后提出一种新的组合爆炸问题的解决方法.

3.1 设计模式检测框架

如图 2 流程所示(图中某些概念见第 3.2 节),设计模式检测方法分为两个阶段:第 1 个阶段是模型训练阶段,第 2 个阶段是模式识别阶段.基本思想是,结合度量和微结构进行设计模式检测.通过为每种设计模式找到合适的度量分类器和微结构分类器,进而训练出堆叠分类器,然后用这些分类器对一个候选的模式实例进行分类,从而预测候选的模式实例是不是真正的模式实例.

设计模式检测相关的度量和微结构是设计模式的两个独立视图,目前还无人提出同时考虑这两个视图的检测方法.另外,因为每个视图最适合的分类器可能不同,如果简单地将两个视图的特征混合到一起之后再训练出一个分类器,那么就有可能丢失这种差异性.因此,本文针对每个视图单独训练分类器,然后再进行模型堆叠.

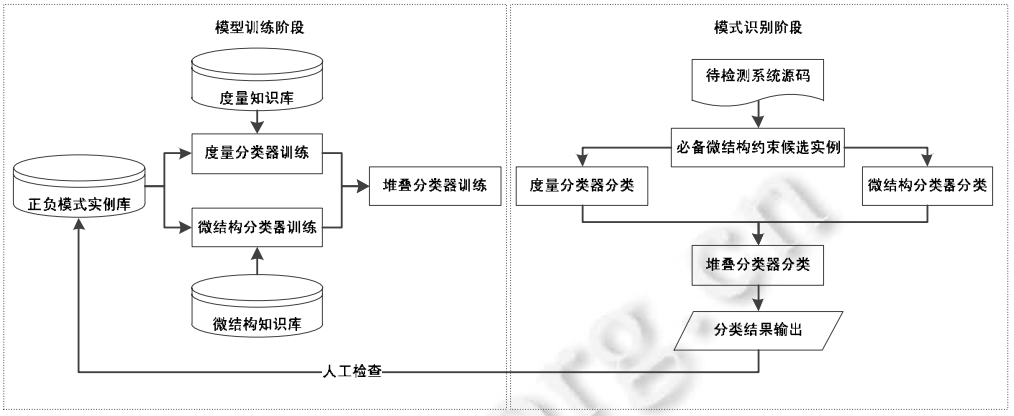


Fig.2 Design pattern detection method based on metrics and micro-structures

图 2 度量和微结构相结合的设计模式检测方法

3.2 基本概念定义

为了方便描述模型训练算法和模式检测算法,本文给出如下符号和定义:

设计模式(design pattern,简称 DP). DP 是 23 种基本的设计模式之一或未来出现的其他设计模式.后续算法将采用此标记.

定义 9(角色映射(role map),简称 RM). 角色映射是一个设计模式角色和角色类之间的对应关系,可以表示为二元组 *RM*=(*Role*,*Class*).

举例来讲,对于 QuickUML2001 项目中存在的一个工厂方法模式实例,存在以下 4 个角色映射:

(Creator,diagram.tool.LinkTool),

(ConcreteCreator,uml.ui.DependencyTool),

(Product,diagram.Link),

(ConcreteProduct,uml.diagram.DependencyLink).

定义 10(模式实例(pattern instance),简称 PI). 模式实例包括角色映射集合 *RoleMaps* 和该实例的类别标签 *Label*,可以表示为 *PI*=(*RoleMaps*,*Label*).其中,*Label*∈{*CORRECT*,*INCORRECT*}.*Label*=*CORRECT* 表示该实例是一个正确的模式实例,或称为正实例;*Label*=*INCORRECT* 表示该实例不是一个正确的模式实例,或称为负实例.

定义 11(正负模式实例库(positive and negative pattern instances repository),简称 PNPIR). 正负模式实例库包括该实例库对应的设计模式 *DP* 和该设计模式的正负实例的集合,可以表示为二元组:

PNPIR=(*DP*,*Instances*).

定义 12(度量知识库(metrics repository),简称 MR). 度量知识库是一个面向对象度量的集合.

举例来讲,本文采用的度量知识库 *MR*={ACAIC,ACMIC,AID,…,connectivity}(完整列表见表 1).需要说明的是,度量知识库并不是一成不变的,可以根据模式检测的需要进行不断地更新.

定义 13(微结构知识库(micro-structure repository),简称 MSR). 微结构知识库是一个设计模式微结构的集合.

举例来讲,本文采用的微结构知识库 *MSR*={Overriding Method,Create Object,Return Type,…,Static Self Instance Returned}(完整列表见表 2).需要说明的是,之所以选择这么少的微结构,是因为本文想先在 5 种设计模式上进行实验,以验证所提方法的有效性.未来为了检测其他的设计模式,这个微结构知识库肯定会包括更多的内容.

定义 14(设计模式度量特征(design pattern metric feature),简称 DPMF). 设计模式度量特征是一个三元组,可以表示为 *DPMF*=(*Role*,*Metric*,*Value*).其中:*Role* 是设计模式的某个角色;*Metric* 是度量知识库中的某个度

量;*Value* 是该度量特征的具体数值,它等于角色 *Role* 对应的角色类的 *Metric* 度量值.

定义 15(设计模式微结构特征(design pattern micro-structure feature),简称 DPMSF). 设计模式微结构特征是一个四元组,可以表示为 *DPMSF*=(*Role*1,*Role*2,*Microstructure*,*Value*).其中:*Role*1 和 *Role*2 是设计模式的的两个角色;*Microstructure* 是微结构知识库中的某个微结构;*Value* 是该微结构特征的具体数值,它等于 *Role*1 对应的角色类→*Role*2 对应的角色类的 *Microstructure* 微结构的数目.

另外,本文是在 Weka 平台上进行实验的,用到的特征选择算法有 CfsSubsetEval,CorrelationAttributeEval, InfoGainAttributeEval,PrincipalComponents 和 ReliefFAttributeEval,用一个集合来表示:*FSASet*={CfsSubsetEval, CorrelationAttributeEval,InfoGainAttributeEval,PrincipalComponents,ReliefFAttributeEval}.

同时,采用 RandomForest,LibSVM,J48 等 15 种常见的有监督学习算法,用一个集合来表示:

CASet={RandomForest,LibSVM,J48,AdaBoostM1,IBk,SMO,LogitBoost,JRip,RandomCommittee, SimpleLogistic,NaiveBayes,KStar,SGD,DecisionTable,ZeroR}.

3.3 3个模型训练算法

在模型训练阶段,相关分类器的训练算法可以描述如下.

度量分类器训练算法以度量知识库、某种设计模式以及相应的正负模式实例库为输入,通过为实例库中的每个实例计算所有的度量特征,从而生成该设计模式的度量分类器的训练数据集.最后,通过遍历特征选择算法和分类算法找到最适合该设计模式的度量特征集和度量分类器.具体算法见表 3.

Table 3 Metric classifier train algorithm

表 3 度量分类器训练算法

Algorithm 1. Metric Classifier Train Algorithm for Every Design Pattern.

Input:度量知识库 *mr*,设计模式 *dp* 以及相应的正负模式实例库 *pnpir*;
Output:设计模式 *dp* 对应的度量特征集 *featureset* 和度量分类器 *classifier*.

```
1  D:=∅
2  foreach instance∈pnpir.Instances do
3      X:=ϕ
4      foreach rolemap∈instance.RoleMaps do
5          foreach metric∈mr do
6              Xi∈DPMF, Xi.Role:=rolemap.Role, Xi.Metric:=metric
7              Xi.Value:=CMeFV(rolemap.Class,metric)
8              X:=X∪{Xi}
9          end
10     end
11     Y:=instance.Label
12     D:=D∪{(X,Y)}
13 end
14 pairs:=FSASet×CASet, featureset:=ϕ, classifier:=ϕ
15 while ∃(algo1,algo2)∈pairs∧¬∃(algo3,algo4)∈pairs(f1Measureof(algo4(D,algo3(D)))>f1Measureof(algo2(D,algo1(D)))) do
16     featureset:=algo1(D)
17     classifier:=algo2(D,featureset)
18     break
19 end
20 return (featureset,classifier)
```

本文为设计模式 *dp* 的所有角色类计算所有的度量值.假设设计模式 *dp* 有 *n* 个角色,度量知识库 *mr* 有 *m* 个度量,那么训练数据集就有 *n*×*m* 个特征,外加一个分类标签.CMeFV 以设计模式的角色类和度量为输入,计算该角色类的相应度量特征值.第 15 行的循环条件表示遍历 FSASet 和 CASet,从而找到使得分类效果最好的特征选择算法和有监督学习算法.其中,algo1 和 algo3 分别表示某种特征选择算法,它们以训练数据集为输入,通过在训练数据集上进行特征选择,输出选择的特征子集;algo2 和 algo4 分别表示某种分类算法,它们以训练数据集和特征子集为输入,通过训练,输出相应的分类器.

*f*1*Measureof*(*)表示括号内分类器在 CORRECT 类别上的 *F*1-*Measure* 值.

微结构分类器训练算法以微结构知识库、某种设计模式以及相应的正负模式实例库为输入,通过为实例库

中的每个实例计算所有的微结构特征,从而生成该设计模式的微结构分类器的训练数据集.最后,同样通过遍历特征选择算法和分类算法找到最适合该设计模式的微结构特征集和微结构分类器.具体算法见表 4.

Table 4 Micro-structure classifier train algorithm

表 4 微结构分类器训练算法

Algorithm 2. Micro-structure Classifier Train Algorithm for Every Design Pattern.

Input:微结构知识库 *msr*,设计模式 *dp* 以及相应的正负模式实例库 *pnpir*;
Output:设计模式 *dp* 对应的微结构特征集 featureset 和微结构分类器 *classifier*.

```
1   D:=ϕ
2   foreach instance∈pnpir.Instances do
3       X:=ϕ
4       foreach rolemap1∈instance.RoleMaps do
5           foreach rolemap2∈instance.RoleMaps do
6               foreach microstructure∈msr do
7                   Xi∈DPMSF, Xi.Role1:=rolemap1.Role
8                   Xi.Role2:=rolemap2.Role
9                   Xi.Microstructure:=microstructure
10                  Xi.Value:=CMsFV(rolemap1.Class,rolemap2.Class,microstructure)
11                  X:=X∪{Xi}
12              end
13          end
14      end
15      Y:=instance.Label
16      D:=D∪{(X,Y)}
17  end
18  ...
19  return (featureset,classifier)
```

CMsFV 以设计模式的两个角色类和微结构为输入,计算这两个角色类的相应微结构特征值.第 18 行省略的部分同表 3 的第 14 行~第 19 行.

堆叠分类器训练算法以某种设计模式以及相应的正负模式实例库、度量特征集、度量分类器、微结构特征集和微结构分类器为输入,通过为实例库中的每个实例计算相关的度量特征和微结构特征,从而生成该设计模式的堆叠分类器的初始训练数据集;然后,通过十折交叉的方法计算每个样本的度量分类器预测值和微结构分类器预测值,并将这两个值作为新的特征添加到样本当中;最后,通过遍历分类算法找到最适合该设计模式的堆叠分类器.具体算法见表 5.

Table 5 Stacked classifier train algorithm

表 5 堆叠分类器训练算法

Algorithm 3. Stacked Classifier Train Algorithm for Every Design Pattern.

Input:设计模式 *dp* 以及相应的正负模式实例库 *pnpir*,度量特征集 *mefeatureset*,
度量分类器 *meclassifier*,微结构特征集 *msfeatureset*,微结构分类器 *msclassifier*;
Output:设计模式 *dp* 对应的堆叠分类器 *classifier*.

```
1   D:=ϕ
2   foreach instance∈pnpir.Instances do
3       X:=ϕ
4       foreach mefeature∈mefeatureset do
5           Xi:=CMeF(instance,mefeature)
6           X:=X∪{Xi}
7       end
8       foreach msfeature∈msfeatureset do
9           Xi:=CMsF(instance,msfeature)
10          X:=X∪{Xi}
11      end
12      Y:=instance.Label
13      D:= D∪{(X,Y)}
14  end
15  subsets:=RandDiv(10,D)
```

Table 5 Stacked classifier train algorithm (Continued)

表 5 堆叠分类器训练算法(续)

Algorithm 3. Stacked Classifier Train Algorithm for Every Design Pattern.

```
16  foreach subset∈subsets do
17      classifier1:=ReTrainClassifier(mefeatureset,meclassifier,{(X,Y)|(X,Y)∈D∧(X,Y)∉subset})
18      foreach sample(X,Y)∈subset do
19          mep:=classify(classifier1,X)
20          X:=X∪{mep}
21      end
22      classifier2:=ReTrainClassifier(msfeatureset,msclassifier,{(X,Y)|(X,Y)∈D∧(X,Y)∉subset})
23      foreach sample(X,Y)∈subset do
24          msp:=classify(classifier2,X)
25          X:=X∪{msp}
26      end
27  end
28  ...
29  return classifier
```

CMeF 以设计模式的实例和度量特征为输入,计算相应的设计模式度量特征;CMsF 以设计模式的实例和微结构特征为输入,计算相应的设计模式微结构特征;RandDiv 以折数 10 和训练数据集为输入,输出随机划分的 10 个训练数据子集;ReTrainClassifier 以度量/微结构特征集、度量/微结构分类器以及重新训练用的数据集为输入,利用度量/微结构分类器的分类算法重新训练分类器;classify 以某个分类器和将要分类的样本的特征向量为输入,输出相应样本的类别标签,也即 CORRECT 或 INCORRECT.第 28 行省略的部分代表遍历 CASet,从而找到最适合设计模式 *dp* 的堆叠分类器 *classifier*.

3.4 设计模式检测算法

在模式识别阶段,本文的设计模式检测算法可以描述如下.

设计模式检测算法以待检测系统的所有类(包括接口)、待检测的设计模式以及相应的度量特征集、度量分类器、微结构特征集、微结构分类器、堆叠分类器为输入,首先计算系统所有类的度量信息和微结构信息,然后生成候选实例,接着对每个候选实例计算度量特征和微结构特征,然后分别用度量分类器和微结构分类器进行分类,最后将两个分类器的预测结果作为新的特征再用堆叠分类器进行分类,从而预测候选实例是不是真正的模式实例.具体算法见表 6.

CalcMe 以待检测系统的所有类和设计模式的度量特征集为输入,为每个类计算度量特征集中所涉及到的所有度量,输出待检测系统的度量信息,它可以看作是类度量表示(CMR)的集合;DetectMs 以待检测系统的所有类和设计模式的微结构特征集为输入,检测微结构特征集中所涉及到的所有微结构,输出待检测系统的微结构信息,它可以看作是微结构的集合;GenCandIns 以待检测系统的所有类以及待检测的设计模式为输入,通过执行本文的候选实例生成算法,输出待检测系统中相应设计模式的候选实例的集合;CMeF2/CMsF2 以候选实例、将要计算的度量特征/微结构特征以及待检测系统的度量信息/微结构信息为输入,通过简单检索,输出相应候选实例的相应度量特征/微结构特征.

需要说明的是:为了避免组合爆炸,本文的候选实例生成算法在考虑设计模式继承层次信息的基础之上加入了一些自己的考量,它的过程可以简单描述如下:(1) 从待检测系统中检测出所有的继承层次;(2) 根据设计模式所包含的继承层次信息,从待检测系统中枚举出候选模式实例;(3) 过滤掉那些不包含设计模式之必备微结构的候选实例(模式必备微结构是指正确模式实例必须具备的微结构,它是候选实例被评估为正确实例的必要非充分条件).本文是依据 GoF 等人给出的定义以及经验数据来定义模式必备微结构的,不会对设计模式检测的正确率与召回率产生影响.举例来讲,对于工厂方法模式,GoF 等人给出的定义是,“定义一个用于创建对象的接口,让子类决定实例化哪一个类.”[1].显然,ConcreteCreator 对 Creator 的 Overriding Method 微结构和 ConcreteCreator 对 Product 类别的 Create Object 微结构就是两个必要的微结构,因为它们是工厂方法模式定义的隐含信息.

Table 6 Design pattern detection algorithm

表 6 设计模式检测算法

Algorithm 4. Design Pattern Detection Algorithm for Every Design Pattern

Input:待检测系统的所有类 *classes*(包括接口),待检测设计模式 *dp* 相应的度量特征集 *mefeatureset*,度量分类器 *meclassifier*,微结构特征集 *msfeatureset*,微结构分类器 *msclassifier*,堆叠分类器 *stackedclassifier*;

Output:待检测系统中所有设计模式 *dp* 的实例集合 *correctins*.

```
1   MeI:=CalcMe(classes,mefeatureset)
2   MsI:=DetectMs(classes,msfeatureset)
3   candins:=GenCandIns(classes,dp)
4   correctins:=ϕ
5   foreach instance∈candins do
6   X:=ϕ
7       foreach mefeature∈mefeatureset do
8           Xi:=CMeF2(instance,mefeature,MeI)
9           X:=X∪{Xi}
10      end
11      foreach msfeature∈msfeatureset do
12      Xi:=CMsF2(instance,msfeature,MsI)
13          X:=X∪{Xi}
14      end
15      mep:=classify(meclassifier,X)
16      X:=X∪{mep}
17      msp:=classify(msclassifier,X)
18      X:=X∪{msp}
19      label:=classify(stackedclassifier,X)
20      if label=CORRECT then
21          correctins:=correctins∪{instance}
22      end
23  end
24  return correctins
```

以上便是本文所提方法的全部.为了丰富正负模式实例库,可以对堆叠分类器的分类结果进行人工检测,并把相应的实例添加到正负模式实例库中.

4 实验评估

为了评估所提方法的有效性,本文进行了相关实验:在模型训练阶段,为实验的每个设计模式找到了合适的度量特征集和微结构特征集,并在此基础上训练出了度量分类器、微结构分类器和堆叠分类器;在模式识别阶段,在 7 个开源项目上进行了实验,并且和现有的两种工具进行了对比分析.

4.1 模型训练评估

目前可用的度量和微结构有很多,可以使用的分类算法也有很多.对于设计模式检测这一问题,本文的思路是:尽可能生成更多的候选度量特征和候选微结构特征,然后再进行特征选择和分类算法选择,通过遍历特征选择算法和分类算法,从而为每种设计模式找到合适的特征集和分类器.现阶段可用的模式实例库主要包括 P-MARt[33],DPB[34]和 Percerons[35]等,本文通过抽取 MARPLE 项目[16]中的训练样本和网络搜集的一些训练样本构造正负模式实例库.具体情况见表 7.

Table 7 PNPIR used in this paper

表 7 本文用到的正负模式实例库的情况

设计模式	正的模式实例数	负的模式实例数	正负模式实例总数
工厂方法模式	139	291	430
单例模式	59	91	150
对象适配器模式	249	288	537
组合模式	30	23	53
观察者模式	60	45	105

在表 7 中,正的模式实例数是指正确的模式实例个数,即正样本的个数;负的模式实例数是指不正确的模式

实例个数,即负样本的个数.对于设计模式检测来讲,什么样的正负样本比例最合适,并没有先验的知识.本文正负模式实例的比例初步控制在 1:3 以内.

下面以观察者模式为例,介绍特征选择和算法选择的结果.观察者模式的类图如图 3 所示.

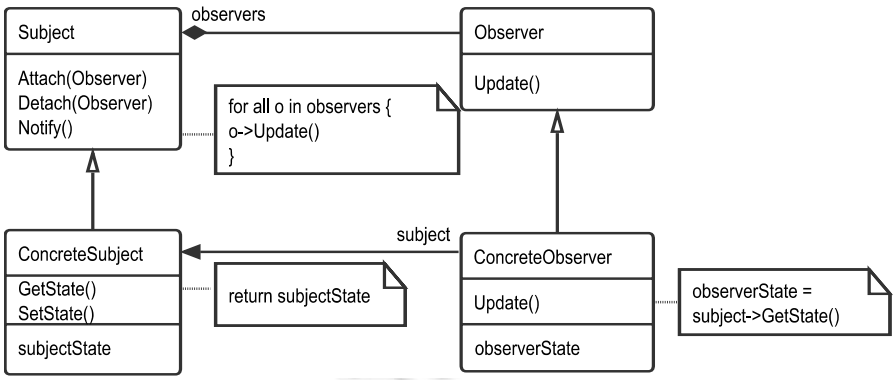


Fig.3 Observer pattern[1]

图 3 观察者模式[1]

从图 3 中可以看出:观察者模式应该有 4 个角色,分别是 Subject,ConcreteSubject,Observer,ConcreteObserver.但是在实际应用中,经常存在 ConcreteSubject/ConcreteObserver 角色缺失或和相应的父类合并的情况.不过,无论如何总会有一个 Subject 类别的角色和一个 Observer 类别的角色,因此对于观察者模式的检测而言,可以只考虑一个 Subject 类别的角色和一个 Observer 类别的角色,从而如果找到了这样一个组合,那么剩下缺失或合并的角色很容易就能通过检查它们所在的继承层次来发现.

表 8~表 10(只显示前 5 项数据)是观察者模式 3 个分类器的训练结果.

从表 8 可以看出:不进行特征选择,并且使用 RandomForest 算法进行模型训练时,所取得的 F1-Measure 最高.从表 9 可以看出:使用 ReliefFAttributeEval 进行特征选择,并且使用 RandomCommittee 算法进行模型训练时,所取得的 *F*1-Measure 最高.从表 10 可以看出,观察者模式的堆叠分类器应该采用 RandomForest 算法.

Table 8 Training results of observer pattern's metric classifier

表 8 观察者模式度量分类器的训练结果

特征选择算法	最佳分类器的指标			最佳分类器对应的分类算法
	Accuracy	*F*1-Measure	AUC	
Origin(不进行特征选择)	**89.523 8**	**0.909 1**	0.930 6	RandomForest
PrincipalComponents	81.904 8	0.850 4	0.855 6	SimpleLogistic
InfoGainAttributeEval	86.666 7	0.885 2	0.897 0	AdaBoostM1
CfsSubsetEval	87.619 0	0.892 6	0.901 7	LogitBoost
CorrelationAttributeEval	88.571 4	0.901 6	0.927 8	RandomForest
ReliefFAttributeEval	88.571 4	0.900 0	**0.936 1**	RandomCommittee

Table 9 Training results of observer pattern's micro-structure classifier

表 9 观察者模式微结构分类器的训练结果

特征选择算法	最佳分类器的指标			最佳分类器对应的分类算法
	Accuracy	*F*1-Measure	AUC	
Origin(不进行特征选择)	83.809 5	0.852 2	**0.869 3**	RandomCommittee
PrincipalComponents	74.285 7	0.765 2	0.793 5	RandomForest
InfoGainAttributeEval	80.000 0	0.810 8	0.822 0	IBk
CfsSubsetEval	80.000 0	0.807 3	0.770 6	AdaBoostM1
CorrelationAttributeEval	81.904 8	0.840 3	0.856 1	RandomCommittee
ReliefFAttributeEval	**84.761 9**	**0.862 1**	0.863 9	RandomCommittee

Table 10 Training results of observer pattern's stacked classifier

表 10 观察者模式堆叠分类器的训练结果

最佳分类器的指标			最佳分类器对应的分类算法
Accuracy	*F*1-Measure	AUC	
90.476 2	**0.916 7**	**0.931 1**	RandomForest
87.619 0	0.894 3	0.869 4	SMO
86.666 7	0.885 2	0.861 1	SGD
86.666 7	0.883 3	0.906 7	AdaBoostM1
85.714 3	0.876 0	0.919 1	RandomCommittee

表 11 显示了观察者模式的微结构特征列表,它是对观察者的候选微结构特征进行特征选择的结果.另外,因为从表 8 可以看出:在训练度量分类器的时候,不进行特征选择并且使用 RandomForest 算法训练时就已经取得了最好的分类效果,而且好于用 CorrelationAttributeEval 进行特征选择后再训练的结果,所以这里并没有给出观察者的度量特征列表.从表 11 可以看出:对于观察者模式,它的微结构特征集包括 14 个特征(表中的 1 表示相应位置的微结构特征被选中).需要说明的是:在使用 ReliefFAttributeEval 进行特征选择时,本文使用 0 作为 ReliefF 评估值的阈值.另外,从表 11 还可以看出:本文新提出的 5 种方法类微结构中,有 4 种与观察者模式的检测有关.

Table 11 List of micro-structure features of observer pattern

表 11 观察者模式的微结构特征列表

特征	Subject	(Subject,Observer)	(Observer,Subject)
Delegation		1	
MultiNotify		1	
ParameterDependence		1	
SingleSelfInstance	1		
StaticSelfInstance	1		
PrivateSelfInstance	1		
StaticSelfInstanceReturned	1		
CreateObject		1	
ProtectedInstantiation	1		
OverridingMethod		1	
Association			1
MultipleRedirectionsinFamily		1	
ParameterizedNotify		1	1

4.2 模式识别评估

在模式识别阶段:首先,对待检测的系统进行预处理,也即计算待检测系统的度量信息和抽取待检测系统的所有微结构;其次,根据微结构信息生成候选实例;再次,计算候选实例的度量特征和微结构特征;最后,用相应的度量分类器、微结构分类器和堆叠分类器进行分类,从而预测一个候选实例是否为真正的模式实例.

本文首先在 JHotDraw v5.1,JRefactory v2.6.24 和 JUnit v3.7 这 3 个开源项目上进行了实验,正如第 3 节所述,为了避免组合爆炸,本文在考虑设计模式继承层次信息的基础之上,加入了设计模式必备微结构的考量.对于工厂方法模式,具体情况见表 12.

Table 12 Candidate instances of factory method pattern

表 12 工厂方法模式的候选实例的组合情况

项目	总类数	默认	继承层次	继承层次和必备微结构
JHotDraw v5.1	155	555 120 720	322 624	4 301
JRefactory v2.6.24	575	108 175 867 800	381 924	948
JUnit v3.7	94	73 188 024	1 764	49

假设待检测项目有 n 个类,那么对于 k 个角色的设计模式,默认情况下,它总共有 $A(n,k)$个候选实例.其中,$A(n,k)$的计算公式如下:

$$A(n,k)=\frac{n!}{(n-k)!} \tag{1}$$

举例来讲,对于表 12 中 JhotDraw v5.1,它共有 155 个类(不考虑内部匿名类),如果从中产生工厂方法模式(k=4)的候选实例,那么候选实例的个数=A(155,4)=555120720.可以看出:默认情况下,工厂方法模式的候选实例的数量非常大.在考虑工厂方法模式的继承层次信息后,候选实例的数量减少到 322 624,缩减了 99%以上(具体过程见 Tsantalis 等人的论文[7]);如果再考虑工厂方法模式的必备微结构信息(参见第 3.4 节),那么候选实例的数量又会进一步缩减 90%以上(最终只剩下 4 301 个候选实例).

通常,每种设计模式都对应一个堆叠分类器.对于实验的 5 种设计模式,由于它们在结构上各不相同,因此都分别对应一个堆叠分类器(对于结构相同的设计模式,比如 State 模式和 Strategy 模式,未来我们准备让它们共用一个堆叠分类器,训练的方法和前面提到的差不多,稍加修改即可).模式检测的过程也就是将一个个候选模式实例分类为 CORRECT 或 INCORRECT 的过程,因此是二分类问题,对应的混淆矩阵如下.

二分类问题混淆矩阵		分类结果	
		CORRECT	INCORRECT
真实结果	CORRECT	*a* (TP)	*b* (FN)
	INCORRECT	*c* (FP)	*d* (TN)

其中,$a+b+c+d$ 等于候选模式实例的个数.由于本文的目的是检测出正确的模式实例,因此选择 CORRECT 类别的准确率作为第一个度量指标,记准确率 $Precision=a/(a+c)$,相应的召回率 $Recall=a/(a+b)$本应该作为第 2 个度量指标,但是考虑到生成的候选模式实例并不一定涵盖所有正确的模式实例,同时也为了能够在不同的设计模式检测方法之间进行比较(基于机器学习的方法和非基于机器学习的方法),因此第 2 个度量指标召回率 *Recall* 应该等于 a/所有检测方法检测到的正确的模式实例个数.最后一个度量指标:

$$F1\text{-}Measure=2\times Precision\times Recall/(Precision+Recall).$$

根据前面所提出的设计模式检测方法,本文开发了设计模式检测工具 OOSdpd 的原型,并且在 JHotDraw v5.1,JRefactory v2.6.24 和 JUnit v3.7 这 3 个开源项目上进行了实验.之所以选择这些项目,是因为这些项目大量使用了设计模式,特别是 JHotDraw,同时很多设计模式检测方法也都是以这 3 个项目为例进行实验的,这样一来就比较容易对比分析.虽然目前有很多种设计模式检测方法,但是提供检测工具的却很少,本文在 P-MARt[33], SSA[7]和所提出的方法之间进行比较,具体实验数据见表 13~表 15.

Table 13 Experimental data of the JHotDraw project

表 13 JHotDraw 项目的实验数据

JHotDraw v5.1	P-MARt			SSA			本文的方法			总数
	检测出	准确率	召回率	检测出	准确率	召回率	检测出	准确率	召回率	
工厂方法模式	3	100%	37.50%	3	100%	37.50%	6	100%	75%	8
单例模式	2	50%	100%	2	50%	100%	1	100%	100%	1
对象适配器模式	21	100%	38.89%	13	100%	24.07%	36	100%	66.67%	54
组合模式	1	100%	100%	1	100%	100%	1	100%	100%	1
观察者模式	2	100%	50%	3	100%	75%	6	66.67%	100%	4

Table 14 Experimental data of the JRefactory project

表 14 JRefactory 项目的实验数据

JRefactory v2.6.24	P-MARt			SSA			本文的方法			总数
	检测出	准确率	召回率	检测出	准确率	召回率	检测出	准确率	召回率	
工厂方法模式	1	100%	11.11%	3	100%	33.33%	6	100%	66.67%	9
单例模式	2	50%	9.09%	12	91.67%	100%	12	91.67%	100%	11
对象适配器模式	23	100%	33.82%	21	100%	30.88%	45	100%	66.18%	68
组合模式	0	–	–	0	–	–	0	–	–	0
观察者模式	0	–	0%	1	100%	50%	3	66.67%	100%	2

Table 15 Experimental data of the JUnit project

表 15 JUnit 项目的实验数据

JUnit v3.7	P-MARt			SSA			本文的方法			总数
	检测出	准确率	召回率	检测出	准确率	召回率	检测出	准确率	召回率	
工厂方法模式	0	–	0%	1	100%	50%	1	100%	50%	2
单例模式	2	0%	–	0	–	–	0	–	–	0
对象适配器模式	0	–	0%	2	100%	50%	4	100%	100%	4
组合模式	1	100%	100%	1	100%	100%	1	100%	100%	1
观察者模式	3	100%	75%	1	100%	25%	2	100%	50%	4

一般来讲,不同的设计模式检测方法针对同一项目、同一设计模式的检测结果也不同,除了方法上的原因之外,还有一个重要的原因,那就是它们选择的模式出现类型不同[36],也即计数模式实例的方法不同.因此,为了在不同的方法之间进行比较,本文使用如下模式出现类型:FactoryMethod:(Creator,Product),Singleton:(Singleton),Adapter:(Target,Adapter),Composite:(Component,Composite),Observer:(Subject,Observer).另外需要说明的是:P-MARt 和 SSA 这两种检测方法并不是基于机器学习的,它们的准确率 *Precision*=检测出的实例中正确的个数/检测出的实例的总个数,召回率 *Recall*=检测出的实例中正确的个数/所有检测方法检测到的正确的模式实例个数,*F*1-*Measure*=2×*Precision*×*Recall*/(*Precision*+*Recall*).

从表 13~表 15 的实验数据来看,本文的方法多数情况下具有较高的准确率和召回率,其中,总数指的是所有检测方法检测到的正确的模式实例个数.从图 4 可以看出:对于工厂方法模式、单例模式和对象适配器模式,本文的方法具有更高的 *F*1-Measure(该值是 3 个项目的平均值).对于组合模式,3 种方法的检测结果一样,因此它们的 *F*1-Measure 也一样.这是因为实验的项目中,组合模式的实例很少,每个项目最多只有一个,3 种方法都能轻松应对.对于观察者模式,本文的方法的 *F*1-Measure 高于 SSA,但是比 P-MARt 稍低,这主要是因为本文的方法目前尚不支持部分角色类存在于第三方库中的情况.另外,P-MARt 和 SSA 的准确率也并非总是 100%的.举例来讲,对于 JHotDraw v5.1 中的单例模式,P-MARt 和 SSA 都检测出了两个单例模式实例 CH.ifa.draw.util.Clipboard 和 CH.ifa.draw.util.Iconkit,但是本文的方法只检测到了一个模式实例 CH.ifa.draw.util.Clipboard.实际上,虽然 CH.ifa.draw.util.Iconkit 很像单例模式,但它并不是,因为它有一个访问权限为 public 的构造函数,因此它不符合单例模式的要求.

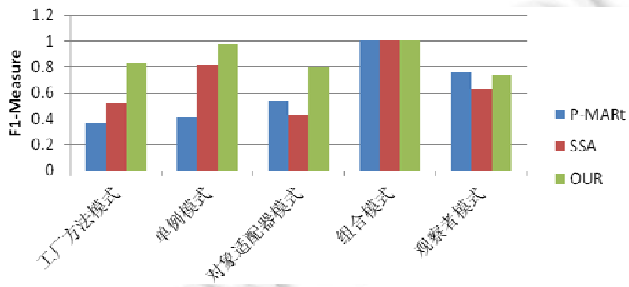


Fig.4 Comparison of detection results on different design patterns

图 4 不同设计模式检测效果对比

图 5 显示了不同检测方法针对不同开源项目的检测效果.需要说明的是:这里的 Average *F*1-*Measure* 代表对于某个开源项目,5 种设计模式检测结果的 *F*1-*Measure* 的平均值.从中可以看出:本文的方法多数情况下检测效果最好,只有在 JUnit v3.7 项目上弱于 P-MARt.其主要原因还是上面提过的,观察者模式的检测效果不如 P-MARt.另外,对于 SSA 和本文的方法,JHotDraw v5.1 的检测效果最好,可能的原因有两个:(1) JHotDraw v5.1 的模式实例占比(模式实例数/项目总类数)更高;(2) JHotDraw 起源于 Gamma 的一个教学实例,而他是软件设计模式概念的提出者之一.最后,从检测效果的稳定性来看,SSA 和本文的方法更加稳定,而 P-MARt 针对不同项目的检测效果差别很大.

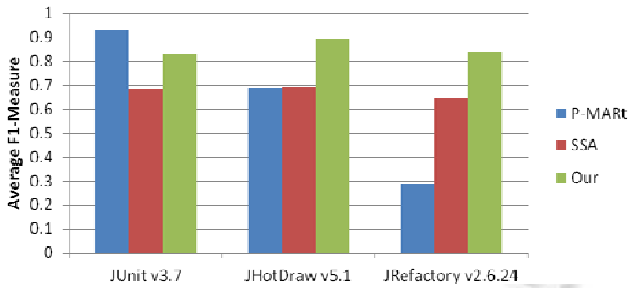


Fig.5 Comparison of detection results on different open source projects

图 5 不同开源项目检测效果对比

上面实验的项目规模都比较小,而且目前大部分设计模式的检测研究都集中在这 3 个项目上.为了验证本文的方法在其他项目上的有效性,以及观察有效性随着项目规模的变化情况,本文又在 ASM v1.4.2(59 个类), Dom4j v1.6.1(189 个类),Guice v3.0(425 个类)和 Struts v2.5.14.1(2 742 个类)等 4 个开源项目上进行了实验.其中, ASM 是常用的 Java 字节码操作框架,Dom4j 是十分优秀的 XML 解析包,Guice 是帮助代码实现控制反转的函数库,Struts 是基于 MVC 的 Web 框架.具体实验数据见表 16~表 19.

由于 P-MARt 实际上是人工标注的实例库,并没有提供可用的检测工具,因此本文的方法只和 SSA 工具进行对比.需要说明的是:为了统计方便,对象适配器模式采用了 Adapter:(Adapter)模式出现类型.

Table 16 Experimental data of the ASM project

表 16 ASM 项目的实验数据

ASM v1.4.2	SSA			本文的方法			总数
	检测出	准确率	召回率	检测出	准确率	召回率	
工厂方法模式	2	100%	100%	2	100%	100%	2
单例模式	1	100%	100%	1	0%	0%	1
对象适配器模式	5	100%	62.50%	6	100%	75%	8
组合模式	0	–	–	0	–	–	0
观察者模式	0	–	–	0	–	–	0

Table 17 Experimental data of the Dom4j project

表 17 Dom4j 项目的实验数据

Dom4j v1.6.1	SSA			本文的方法			总数
	检测出	准确率	召回率	检测出	准确率	召回率	
工厂方法模式	0	–	0.00%	14	78.57%	100.00%	11
单例模式	8	0%	–	5	0%	–	0
对象适配器模式	7	100%	46.67%	13	100%	86.67%	15
组合模式	0	–	0%	3	100%	100%	3
观察者模式	0	–	0%	1	100%	100%	1

Table 18 Experimental data of the Guice project

表 18 Guice 项目的实验数据

Guice v3.0	SSA			本文的方法			总数
	检测出	准确率	召回率	检测出	准确率	召回率	
工厂方法模式	14	92.86%	92.86%	2	50%	7.14%	14
单例模式	9	44.44%	80%	2	100%	40%	5
对象适配器模式	26	100%	89.66%	18	100%	62.07%	29
组合模式	0	–	0%	2	100%	100%	2
观察者模式	0	–	0%	1	100%	100%	1

Table 19 Experimental data of the Struts project

表 19 Struts 项目的实验数据

Struts v2.5.14.1	SSA			本文的方法			总数
	检测出	准确率	召回率	检测出	准确率	召回率	
工厂方法模式	11	72.73%	53.33%	16	93.75%	100%	15
单例模式	8	25%	12.50%	20	70%	87.50%	16
对象适配器模式	65	100%	43.92%	140	100%	94.59%	148
组合模式	1	100%	20%	6	83.33%	100%	5
观察者模式	4	75%	42.86%	7	71.43%	71.43%	7

从表 16 可以看出,本文的方法只在单例模式上弱于 SSA.从表 17 可以看出,本文的方法在实验的 5 种设计模式上均好于 SSA.从表 18 可以看出,本文的方法在工厂方法模式上明显弱于 SSA.这主要是因为 Guice v3.0 项目中包含了很多内部类实现的工厂方法模式,而本文的方法用的 POM Tests 度量计算工具并不支持内部类,因此许多实例没有检测出来;另外,对于单例模式,和 SSA 相比,本文的方法准确率高,但是召回率低,不过两者的 *F*1-*Measure* 非常接近.从表 19 可以看出,本文的方法在所有模式上均好于 SSA.

图 6 显示的是每种设计模式的平均检测效果,从中可以看出:本文的方法只在单例模式上弱于 SSA,但是差距并不明显;在组合模式和观察者模式上,本文的方法明显好于 SSA,这主要是因为本文的方法使用了更多能够体现模式结构和行为特征的微结构信息.图 7 显示的是针对不同项目规模,SSA 和本文的方法在 5 种设计模式上的平均检测效果对比,从中可以看出:对于不同规模的项目,本文方法的检测效果和 SSA 的检测效果都有一定的波动,但是本文方法的检测效果更加稳定.另外,对于规模比较小的项目(ASM v1.4.2 和 Guice v3.0),本文的方法和 SSA 相比优势并不明显;但是对于规模比较大的项目(Struts v2.5.14.1),本文的方法要明显好于 SSA.因此,从实验数据来看,本文的方法更加适合规模比较大的项目.

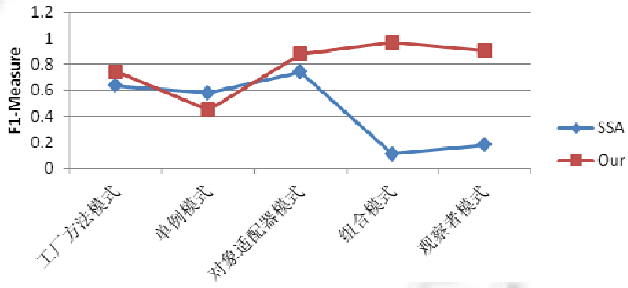


Fig.6 Comparison of detection results on different design patterns

图 6 不同设计模式检测效果对比

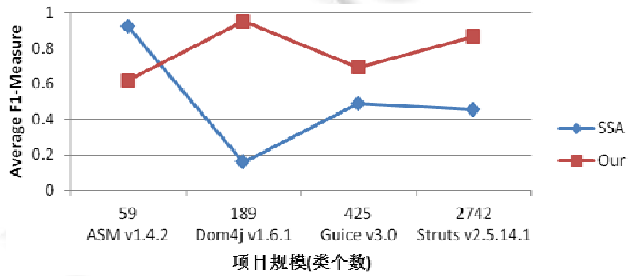


Fig.7 Comparison of detection results on different project scale

图 7 不同项目规模检测效果对比

通过以上两组对比实验可以看出,本文的方法和 P-MARt 以及 SSA 相比,多数情况下具有更高的识别准确率和召回率.对于单例模式,本文的方法还需要进一步改进.对于观察者模式,本文的方法的检测效果要明显好

于单纯基于结构相似度计算的 SSA.

虽然目前有很多种基于机器学习的设计模式检测方法,但是我们没能找到任何一款可用的检测工具,而且这些方法多数也没有指明所采用的模式出现类型,因此很难以拿本文的方法与它们进行比较.

4.3 有效性分析

影响外部有效性的因素主要有以下 3 点.

1) 与其他基于机器学习的检测方法一样,本文的训练数据集也是由人工产生的,这就会受到主观因素的影响.因此,需要找到一个广泛认可的设计模式实例库作为训练数据集的来源;
2) 本文以工厂方法模式为例,定义了它的两个必备微结构,从而显著地缩减了候选实例的搜索空间,避免了组合爆炸.但是对于其他设计模式,定义模式必备微结构对避免组合爆炸的有效程度还需要进一步研究;
3) 虽然本文在观察者模式上取得了更好的检测效果,但是对于其他行为型设计模式的检测效果还需要进一步验证.

影响内部有效性的因素主要有以下两点:(1) 由于训练样本不够丰富以及所采用的度量知识库和微结构知识库不够完备,本文提出的方法对单例模式的检测效果还不够理想,不足以验证所提方法的有效性;(2) 如果模式实例中含有部分存在于第三方库中角色类,本文的方法无法准确检测,需要进一步研究如何将第三方库中的有效信息加载到检测目标项目中.

5 总结与展望

针对目前设计模式检测方法的研究中存在的若干问题,本文提出了一种基于机器学习的设计模式检测方法.首先,针对模式实例变体检测效果不理想的问题,本文利用面向对象度量特征和微结构特征进行基于堆叠泛化的学习,实验结果表明,对变体的检测效果有较大改善;其次,针对行为型设计模式的检测复杂和难以实现的问题,本文从微结构的角度追踪行为型设计模式的行为特征,对于观察者模式,相较于 SSA 等经典方法取得了更高的 *F*1-*Measure* 值;再次,针对设计模式检测中的发生的组合爆炸问题,本文在考虑设计模式继承层次信息的基础之上,加入特定的设计模式微结构的考量,从而进一步缩减了模式的搜索空间.

具体地讲,本文采用机器学习的方式分别从度量和微结构两个视图训练分类器,然后采用模型堆叠的方式训练出最终的分类器.在几个开源项目上的实验表明:本文的方法相对于 P-MARt 和 SSA 来讲,具有更好的模式检测效果.

在未来工作中,本文将从如下几个方面进行改进和扩展.

(1) 本文应用比特向量表示方法类微结构的位置和数目,并通过比特向量及其位运算识别设计模式的方法映射.在未来工作中我们会将这些位置信息也输入分类器,更充分地利用微结构的位置信息,从而更加准确地追踪设计模式的行为特征;
(2) 找到更为广泛认可的设计模式实例库作为训练数据集的来源,从而提高设计模式检测的准确率;
(3) 进一步研究将第三方库中的有效信息加载到检测目标项目中的方法并完善度量知识库和微结构知识库,从而提高设计模式检测方法的可用性.

References:

[1] Gamma E. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1995.
[2] Firesmith DG, Eykholt EM, Siegel J. Dictionary of Object Technology: The Definitive Desk Reference. Cambridge University Press, 1995.
[3] Mohri M, Rostamizadeh A, Talwalkar A. Foundations of Machine Learning. MIT Press, 2012.
[4] Heuzeroth D, Holl T, Hogstrom G, *et al*. Automatic design pattern detection. In: Proc. of the 11th IEEE Int'l Workshop on Program Comprehension. IEEE, 2003. 94–103.

[5] Wendehals L. Improving design pattern instance recognition by dynamic analysis. In: Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA). 2003. 29−32.

[6] De Lucia A, Deufemia V, Gravino C, *et al*. Towards automating dynamic analysis for behavioral design pattern detection. In: Proc. of the 2015 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). IEEE, 2015. 161−170.

[7] Tsantalis N, Chatzigeorgiou A, Stephanides G, *et al*. Design pattern detection using similarity scoring. IEEE Trans. on Software Engineering, 2006,32(11):896−909.

[8] Dong J, Sun Y, Zhao Y. Design pattern detection by template matching. In: Proc. of the 2008 ACM Symp. on Applied Computing. ACM, 2008. 765−769.

[9] Yu D, Zhang Y, Chen Z. A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. Journal of Systems and Software, 2015,103:1−16.

[10] Oruc M, Akal F, Sever H. Detecting design patterns in object-oriented design models by using a graph mining approach. In: Proc. of the 2016 4th Int'l Conf. on Software Engineering Research and Innovation (CONISOFT). IEEE, 2016. 115−121.

[11] Mayvan BB, Rasoolzadegan A. Design pattern detection based on the graph theory. Knowledge-Based Systems, 2017,120:211−225.

[12] Dietrich J, Elgar C. Towards a Web of patterns. Web Semantics: Science, Services and Agents on the World Wide Web, 2007,5(2): 108−116.

[13] Kirasić D, Basch D. Ontology-Based design pattern recognition. In: Proc. of the Int'l Conf. on Knowledge-Based and Intelligent Information and Engineering Systems. Berlin, Heidelberg: Springer-Verlag, 2008. 384−393.

[14] Panich A, Vatanawood W. Detection of design patterns from class diagram and sequence diagrams using ontology. In: Proc. of the 2016 IEEE/ACIS 15th Int'l Conf. on Computer and Information Science (ICIS). IEEE, 2016. 1−6.

[15] Alhusain S, Coupland S, John R, *et al*. Towards machine learning based design pattern recognition. In: Proc. of the 2013 13th UK Workshop on Computational Intelligence (UKCI). IEEE, 2013. 244−251.

[16] Zanoni M, Fontana FA, Stella F. On applying machine learning techniques for design pattern detection. Journal of Systems and Software, 2015,103:102−117.

[17] Chihada A, Jalili S, Hasheminejad SMH, *et al*. Source code and design conformance, design pattern detection from source code by classification approach. Applied Soft Computing, 2015,26:357−367.

[18] Zanoni M. Data mining techniques for design pattern detection. Università degli Studi di Milano-Bicocca, 2012.

[19] Finder [design pattern detection]. https://wiki.cse.yorku.ca/project/dpd/finder

[20] Antoniol G, Fiutem R, Cristoforetti L. Using metrics to identify design patterns in object-oriented software. In: Proc. of the 5th Int'l Software Metrics Symp. IEEE, 1998. 23−34.

[21] Guéhéneuc YG, Sahraoui H, Zaidi F. Fingerprinting design patterns. In: Proc. of the 11th Working Conf. on Reverse Engineering. IEEE, 2004. 172−181.

[22] Issaoui I, Bouassida N, Ben-Abdallah H. Predicting the existence of design patterns based on semantics and metrics. The Int'l Arab Journal of Information Technology, 2016,13(2):310−319.

[23] Chidamber SR, Kemerer CF. A metrics suite for object oriented design. IEEE Trans. on Software Engineering, 1994,20(6): 476−493.

[24] Lorenz M, Kidd J. Object-Oriented Software Metrics: A Practical Guide. Prentice-Hall, Inc., 1994.

[25] Tegarden DP, Sheetz SD, Monarchi DE. A software complexity model of object-oriented systems. Decision Support Systems, 1995, 13(3-4):241−262.

[26] Hitz M. Measuring coupling and cohesion in object-oriented systems. In: Proc. of the Int'l Symp. on Applied Corporate Computing. Monterrey, 1995. 25−27.

[27] Briand LC, Daly JW, Wuest J. A unified framework for cohesion measurement. In: Proc. of the 4th Int'l Software Metrics Symp. IEEE, 1997. 43−53.

[28] Nickel U, Niere J, Zündorf A. The FUJABA environment. In: Proc. of the 22nd Int'l Conf. on Software Engineering. ACM, 2000. 742−745.

[29] Smith JM, Stotts D. An elemental design pattern catalog. Technical Report, 02-040, University of North Carolina at Chapel Hill, 2002.

[30] Gil J Y, Maman I. Micro patterns in Java code. ACM SIGPLAN Notices, 2005,40(10):97−116..

[31] Fontana FA, Maggioni S, Raibulet C. Design patterns: A survey on their micro-structures. Journal of Software: Evolution and Process, 2013,25(1):27−52.

[32] Fontana FA, Maggioni S, Raibulet C. Understanding the relevance of micro-structures for design patterns detection. Journal of Systems and Software, 2011,84(12):2334−2347.

[33] Guéhéneuc YG. P-Mart: Pattern-like micro architecture repository. In: Proc. of the 1st EuroPLoP Focus Group on Pattern Repositories. 2007. 1−3.

[34] Fontana FA, Caracciolo A, Zanoni M. DPB: A benchmark for design pattern detection tools. In: Proc. of the 16th European Conf. on Software Maintenance and Reengineering (CSMR). IEEE, 2012. 235−244.

[35] Ampatzoglou A, Michou O, Stamelos I. Building and mining a repository of design pattern instances: Practical and research benefits. Entertainment Computing, 2013,4(2):131−142.

[36] Pettersson N, Lowe W, Nivre J. Evaluation of accuracy in design pattern occurrence detection. IEEE Trans. on Software Engineering, 2010,36(4):575−590.

冯铁(1972－),男,吉林长春人,博士,副教授,CCF 专业会员,主要研究领域为软件维护与演化,逆向工程,程序理解,软件重构,软件自动化.

靳乐(1991－),男,硕士,主要研究领域为软件维护与演化,程序理解,软件重构.

张家晨(1969－),男,博士,教授,CCF 高级会员,主要研究领域为软件维护与演化,软件集成,网构软件,软件工程环境,软件开发辅助工具.

王洪媛(1974－),女,博士,讲师,主要研究领域为需求工程,软件建模,体系结构,上下文信息管理.